Chapitre 5 : Les sous-algorithmes (Procédures et fonctions)

Comment faire pour réutiliser un même calcul dans plusieurs algorithmes sans le réécrire à chaque fois ? Peut-on demander à un algorithme d'exécuter un traitement précis, mais seulement quand c'est nécessaire, sans toucher au reste du programme ? Quelles erreurs peut-on éviter en utilisant des sous-algorithmes plutôt qu'un seul gros algorithme ?

Ce chapitre répondra à ces questions.

5.1 Introduction

Lorsqu'un algorithme devient complexe, il peut contenir de nombreuses instructions répétitives ou des traitements qui se répètent à plusieurs endroits. Écrire plusieurs fois les mêmes instructions rend le programme long, difficile à lire et à maintenir.

Pour résoudre ce problème, on utilise les sous-algorithmes, c'est-à-dire des parties d'algorithme que l'on peut isoler, nommer et réutiliser à volonté. Un sous-algorithme peut recevoir des informations (paramètres), effectuer un traitement précis, puis retourner un résultat.

5.2 Notion de sous-algorithme

Un sous-algorithme est un algorithme à l'intérieur d'un autre algorithme. Il possède la même structure que l'algorithme principal. Il peut être appelé par l'algorithme principal ou par un autre sous-algorithme pour réaliser un certain traitement et retourner des résultats (un ou plusieurs résultats).

5.3 Déclaration d'un sous-algorithme

La déclaration d'un sous-algorithme consiste à préciser sa structure avant son utilisation dans un algorithme principal.

Un sous-algorithme se déclare dans la partie déclarative de l'algorithme principal, juste après la déclaration des variables et constantes.

Algorithme id_Algorithme ;

Variables <Déclaration des variables>;
Constantes <Déclaration des constantes>;
<Déclaration des sous-algorithmes> ;

Début
<instruction_1> ;
.
<Appels des sous-algorithmes> ;
.
<instruction_N> ;

Fin.

En C :

```
#include <stdio.h>

<Déclaration des variables>;
<Déclaration des constantes>;
<Déclaration des sous-programmes> ;

int main() {

    <instruction_1> ;
    <Appels des sous-programmes> ;

    <instruction_N> ;

     return 0;
}
```

Remarque : En langage C, les sous-programmes sont déclarées avant la fonction main, afin de permettre leur appel à l'intérieur du programme principal.

Il existe deux types deux sous-algorithmes : Les fonctions et les procédures.

5.3.1 Déclaration d'une fonction

Une fonction est un type de sous-algorithme qui permet de retourner une valeur unique, elle possède, donc, un type de retour. Une fonction est déclarée en suivant la structure suivante :

Fonction id_fonction (P1 : Type1 ; P2 : Type2 ; … ; PN : TypeN) : Type_Fonction ;
Variables <Déclaration des variables ;>
Début
Instruction(s) ;
Retourner Resultat ;
Fin ;

id_fonction est l'identificateur de la fonction.

P1, P2, …, PN représentent les paramètres formels, chacun est associé à un type, permettant de transmettre des données à la fonction.

Type_Fonction désigne le type de la valeur retournée par la fonction.

La partie ***Variables*** contient les déclarations locales des variables utilisées par la fonction en plus des paramètres.

L'instruction ***Retourner*** Résultat permet de renvoyer la valeur calculée.

En C :

```
Type_Fonction id_fonction (Type 1 P1, …, TypeN PN) {
  <Déclaration des variables ;>
      Instructions ;
      return Resultat ;
}
```

5.3.2 Déclaration d'une procédure

Une procédure est un type de sous-algorithme qui réalise un traitement donné sans retourner de valeur, contrairement à la fonction. Une procédure ne possède, donc, pas un type de retour. Une procédure est déclarée en suivant la structure suivante :

Procédure id_ procédure (P1 : Type1 ; P2 : Type2 ; … ; PN : TypeN) ;
Variables <Déclaration des variables ;>
Début
  Instruction(s) ;
Fin ;

En C :

```
void id_procedure (Type 1 P1, …, TypeN PN) {
  <Déclaration des variables ;>
     Instructions ;
}
```

5.4 Passage de Paramètres et Portée des Variables

Le passage de paramètres est un mécanisme de communication permettant aux sous-algorithmes d'échanger des informations entre eux et avec l'algorithme principal.

Il existe deux types de passage de paramètres : par valeur et par adresse.

5.4.1 Passage par valeur (Transmission par valeur)

- Une copie de la valeur du paramètre effectif est envoyée au sous-algorithme.
- Le sous-algorithme travaille sur cette copie, pas sur la variable d'origine.
- Tout changement local n'affecte pas la variable dans l'algorithme principal.
- Utilisé pour les paramètres d'entrée.

Remarque : Un paramètre passé par valeur est précédé soit de la lettre E (Entrée), soit de rien du tout ; par défaut, tout paramètre est considéré comme passé par valeur.

5.4.2 Passage par adresse / par référence

- Le sous-algorithme reçoit l'adresse de la variable.
- Toute modification affecte directement la variable réelle (paramètre effectif).
- Utilisé pour les paramètres de sortie ou d'entrée-sortie.
- En algorithmique, on utilise S (Sortie), ou E/S (Entrée-Sortie) devant le paramètre formel passé par adresse.

Remarque : En langage C, lorsqu'un paramètre est passé par adresse, il est précédé par un astérisque * dans la déclaration du sous-algorithme (paramètre formel).

Lors de l'appel, le paramètre effectif correspondant est précédé par une esperluette &.

5.5 Appels des sous-algorithmes

L'appel d'un sous-algorithme permet d'exécuter une fonction ou une procédure à partir de l'algorithme principal ou d'un autre sous-algorithme.

Lors de l'appel, les valeurs transmises sont appelées paramètres effectifs.

Pour que l'appel soit correct, il doit respecter les règles suivantes :

- Le même nombre de paramètres que dans la déclaration,
- Le même ordre des paramètres,
- Le même type pour chaque paramètre correspondant.

Exemple :

Algorithme ExemplePassage ;
Variables y : entier ;
Procédure SA1 (x : entier) ; //Passage par valeur
Début

x ← x + 10;

Fin ;

Procédure SA2 (**E/S** x : entier) ; //Passage par adresse
Début

x ← x + 10;

Fin ;

Début
y ← 5;
SA1(y) ;
Ecrire(y) ; **// Ici y reste 5**
SA2(y) ;
Ecrire(y) ; **// Ici y devient 15**
Fin.

Dans l'exemple, ci-avant, lors de l'appel de SA1, le paramètre est passé par valeur : la modification de x n'affecte pas la variable y, qui conserve donc sa valeur initiale.
En revanche, lors de l'appel de SA2, le paramètre est passé par adresse : toute modification de x agit directement sur y, qui prend alors la nouvelle valeur.

5.6 Portée des variables

Lorsque un algorithme contient des sous-algorithmes (fonctions, procédures), toutes les variables n'ont pas la même « portée », c'est-à-dire la zone où elles sont visibles et utilisables.

5.6.1 Variables globales

Les variables globales sont déclarées avant les sous-algorithmes. Elles sont accessibles partout : dans l'algorithme principal et dans les sous-algorithmes (lecture et modification).

Les variables globales permettent de partager des informations entre le programme principal et les sous-algorithmes.

5.6.2 Variables locales

Les variables locales sont déclarées à l'intérieur d'un sous-algorithme. Elles sont accessibles uniquement dans ce sous-algorithme (elles n'existent pas avant ni après son exécution).

Les variables locales servent pour des calculs internes sans modifier le reste du programme.

Remarques :

1. Lorsqu'une variable locale porte le même identificateur qu'une variable globale, la variable locale masque la globale à l'intérieur du sous-algorithme.

Autrement dit, dans le sous-algorithme, toutes les opérations utilisent la variable locale, et la variable globale est ignorée jusqu'à la fin du sous-algorithme.

2. Les paramètres formels d'un sous-algorithme sont considérés comme des variables locales.

5.7 La récursivité

La récursivité est une méthode naturelle pour programmer des fonctions définies par récurrence, car elle reprend exactement le même principe : un cas simple (Cas trivial) et un appel sur un sous-problème plus petit. Le cas trivial, permet de stopper les appels récursifs et à procéder au renvoi des résultats.

Si l'on considère la suite S = 1+2+3+⋯+N, qui permet de calculer la somme des N premiers nombres, on peut tout d'abord écrire sa fonction itérative comme suit :

```
Fonction Somme(N : entier) : entier ;
Variables S , i: entier ;
Début
    S ← 0 ;
    Pour i ← 1 à N faire
        S ← S + i ;
    Fin-Pour ;
Retourner S ;
Fin ;
```

On remarque ensuite que ce problème est défini par récurrence, car la somme de N nombres peut s'exprimer comme la somme des (N-1) premiers nombres plus N.
On peut donc écrire une fonction récursive :

```
Fonction Somme(N : entier) : entier ;
Variables S : entier ;
Début
  Si (N = 1) alors
     S ← 1; // Cas trivial
  Sinon
     S ← N + Somme(N-1) ;
  Fin-Si ;
Retourner S ;
Fin ;
```

Pour N = 1, la somme des N premiers nombres se réduit à ce seul nombre, donc la somme vaut 1. On connaît ce résultat directement, ce qui en fait le cas trivial.
De plus, ce cas ne nécessite aucun autre appel récursif, ce qui permet d'arrêter la récursion.

Remarques :

Les fonctions récursives peuvent être transformées en procédures récursives et de manière générale, toute procédure peut être transformée en procédure et vice versa.

5.8 Étapes de transformation d'une fonction en procédure et d'une procédure en fonction

Pour transformer une fonction en procédure, on suit les étapes suivantes :

1. Remplacer le mot-clé Fonction par Procédure.
2. Supprimer le type de retour de la fonction.
3. Prendre la variable résultat de la fonction et l'ajouter comme paramètre formel de la procédure, passé par adresse (E/S).
4. Supprimer l'instruction Retourner dans la procédure.

Dans l'algorithme principal, il faut :

- Ajouter une variable qui servira de paramètre effectif correspondant au paramètre formel résultat lors de l'appel de la procédure.
- Si l'appel de la fonction se trouvait dans une affectation, il faut extraire cet appel et l'écrire comme instruction à part entière, puis utiliser la variable ajoutée pour récupérer le résultat.

Pour transformer une procédure en fonction, il suffit de procéder inversement à toutes les étapes précédemment décrites.

5.9 Conclusion

Ce chapitre a présenté les sous-algorithmes, en distinguant fonctions et procédures, ainsi que leurs déclarations et appels.

Ces notions constituent la base pour structurer efficacement un programme et réutiliser des traitements dans différents contextes.